Attorney Docket No.: 09623V-021100

PATENT APPLICATION

MULTI-INSTANCE INPUT DEVICE CONTROL

Inventors:

Aaron Standridge, a citizen of United States, residing at
36289 Casey Court
Newark, CA 94560

Tim Dieckman, a citizen of United States, residing at
470 Warren Drive, #301
San Francisco, CA 94131

Assignee:

Logitech Europe S.A.
Moulin du Choc
CH-1122 Romanel-sur-Morges
Switzerland

Entity: Large

TOWNSEND and TOWNSEND and CREW LLP
Two Embarcadero Center, 8th Floor
San Francisco, California 94111-3834
Tel: 415-576-0200

MULTI-INSTANCE INPUT DEVICE CONTROL

CROSS-REFERENCES TO RELATED APPLICATIONS

This application is a continuation in part ("CIP") of Application No. 09/438,012, filed November 10, 1999, for MULTI INSTANCE INPUT DEVICE CONTROL.

BACKGROUND OF THE INVENTION

The present invention relates to media source input devices such as microphones and video cameras, and in particular to the interfacing of media source input devices to application programs.

Traditionally, when one application program connects to a media source, all other application programs are prevented from using that media source. In the context of a common personal computer, when an application program calls to communicate with a media source, the application program calls to the driver files or the dynamic link library (DLL or *.dll) files. Typically, a DLL provides one or more particular functions and a program accesses the function by creating a link to the DLL. DLL's can also contain data. Some DLL's are provided with the operating system (such as the Windows operating system) and are available for any operating system application. Other DLL's are written for a particular application and are loaded with the application program (such a media source control application program). When a media source control application program calls to connect to a media source, at that point, the driver checks to make sure that no other application has opened the particular camera driver file (*.dll), and if no other has, the driver will open the particular driver file. Having done so, there now exists a single threaded connection between the media source (e.g., video camera) and the application program through the opened media source (e.g., video camera) driver file as seen in Fig. 1.

Fig. 1 shows an application program connecting with a media source, which is a video camera. As depicted in Fig. 1, the driver file 14 is opened by the driver 12 which is called by the calling application program 10 and gets loaded in the calling application program's memory. Since the video camera driver file 14 has been opened by the application program 10, the next application that attempts to make a call to the video camera is prevented from doing so. The issues related to conflicts in sharing a media source between multiple application programs is known as contingency issues. There

will be contingency issues, since typical input device drivers only allow one application to use the input device data at any given time. This is because the video camera driver file has been loaded in the first application program's memory and is not available to be accessed by another calling program. Therefore, each application program that potentially makes calls to a video camera must account for the presence of another application program possibly already using the camera. Accordingly, such application programs are encumbered by the need to first check to determine whether another first application program was executed that had connected to the video camera, and if so the second calling program must have routines allowing it to negotiate the sharing of the camera. However such sharing is a single-instant one, meaning the that connection between the camera and the first application program would have to be broken (i.e. the first application program would have to be shut down or the video camera turned off) before the connection between camera and the second application program could be established. Authority, priority, and other security aspects as well as appropriate error handling must also be resolved by the communications between the two competing application programs. Presently, no application program even attempts to resolve any of these issues, and therefore if a connection between a calling program and a camera cannot be established, the unexpected application programs errors are resolved by the operating system which issues rather inelegant and undecipherable error messages leaving the ultimate user to only infer that a proper connection could not be established. At best, the second calling application program receives a message that the device being called to is currently in use and not available.

Application programs have continued to grow in size, flexibility and usability, and the trend has been to move away from large monolithic application programs to programs that are made of many smaller sub-programs. This building block approach provides many advantages such as ease of later modification and configurability. Moreover, operating system suppliers, such as Microsoft, have also adopted such a modular approach and hence offer many standard sub-programs or objects that handle many utility-type functions such as queuing files to a printer, and loading and running printer driver (e.g., DLL) files to print files. The driver (e.g., DLL) files themselves are objects or sub-programs. Further, in an effort to allow interoperability between objects and smaller sub-programs written in different high level programming languages, operating systems suppliers have developed models for executable programs, which can be compatible with each other at the binary level. One such model for binary

code developed by the Microsoft Corporation is the component object model (COM). The COM enables programmers to develop objects that can be accessed by any COM-compliant application. Although many benefits can be realized by transitioning from large monolithic application programs to sets of smaller sub-programs and objects, those advantages must be balanced against the burdens imposed by the need for the additional routines allowing for inter process communications amongst these sub-programs and objects.

Besides growing in complexity and usability, multi-unit application programs have been migrating from single-host sites to multiple host heterogeneous network environments. Consequently, it is now not unheard of to have a single application program be comprised of many different routines, each written in different high level programming languages and each residing on a separate computer, where all those computers are connected to each other across a network. In such implementations, the demands for efficient intra and inter-network and inter-process communications can take on a life of their own, detracting from the programmer's primary function of writing an application program. The programmer also has to handle the communications issues posed by spreading application programs across a network. Once again, operating systems suppliers have realized this challenge and potential detraction and have addressed it in various ways. For example, Microsoft has extended the COM functionality by developing the distributed component object model (DCOM). DCOM is an extension of COM to support objects distributed across a network. Besides being an extension of COM, DCOM provides an interface that handles the details of network communication protocols allowing application programmers to focus on their primary function of developing application specific programs. DCOM is designed to address the enterprise requirements for distributed component architecture. For example, a business may want to build and deploy a customer order entry application that involves several different areas of functionality such as tax calculation, customer credit verification, inventory management, warranty update and order entry. Using DCOM the application may be built from five separate components and operated on a web server with access via a browser. Each component can reside on a different computer accessing a different database. The programmer can focus on application development and DCOM is there to handle the inter process communications aspects of the separate components of the application program. For example, DCOM would handle the integration of component

communication with appropriate queues and the integration of component applications on a server with HTML-based Internet applications.

Thus, while many computer system operating system suppliers are providing many standardized models for executable programs, even such executable programs can only interface with a media source input device on a one-on-one basis. A standardized device driver file, once linked to an application program, is no longer available for use by another program. There is a need to allow multiple application programs to share a single media source input device, which most commonly is a video camera or microphone.

SUMMARY OF THE INVENTION

The present invention combines features of an executable process with the need for multiple application programs to share a single input device, such as video camera or a microphone. An input device such as a video camera or a microphone is a peripheral device that is opened and remains open in response to a call from an application programs. The present invention provides an executable program implemented as a process that allows multiple applications to communicate with a single input device. This is achieved by creating a virtual interface (an instance) to the physical input device and by loading the input device control executable program into a process. An instance is an actual usage and the resulting virtual creation of a copy of an entity loaded into memory. The executable program process acts as a server thus allowing multiple application programs to interface with the same input device. This executable program, which as used herein is referred to as the multi-instance input device control (MIIDC) executable program responds to each application program request as if the input device is open for the calling application program. Each application program is thus enabled to communicate with the input device instance without interrupting the operation of other application programs communicating with the same input device. In other words, the MIIDC virtualizes an input device by creating s client-server architecture, where each calling application program is a client and where the MIIDC is the server, serving the driver file to each calling application program.

The MIIDC and the method of virtualizing an input device are implementable on many computing platforms running various operating systems. A media source input device such as a video camera or a microphone is commonly interfaced with a host computer. The host computer is most commonly a personal

computer, such as the commonly available PC of Mac computers. However, since advancements in technology are blurring the boundaries between computing and communication devices, a host computer as used herein is synonymous with an intelligent host, and an intelligent host as used herein is meant to include other examples of any host having a processor, memory, means for input and output, and means for storage. Other examples of intelligent hosts, which are also equally qualified to be used in conjunction with embodiments of the present invention include a handheld computer, an interactive set-top box, a thin client computing device, a personal access device, a personal digital assistants, and an internet appliance.

In one implementation on a PC host running a common Windows-based operating system, the (MIIDC) executable program can be a DCOM object. DCOM can also serve as an interface that allows multiple application programs to communicate with a single input device. The DCOM interface handles all interfacing operations such as: loading, executing, buffering, unloading and calling to the executable program. In the DCOM-based implementation, the MIIDC object itself is a DCOM server. The MIIDC program works by connecting to the input device in a DCOM object implemented as an executable server. Consequently, the MIIDC becomes a DCOM object implemented as an executable program, meaning that MIIDC is a process – like any other operating system (O/S) process – sharable by many applications. By placing the input device access program into a separate executable process, the input device is capable of being shared by multiple application programs. The DCOM interface appears to the application program as if it is being opened just for the application that calls to the DCOM object, while there's only one instance of the input device.

MIIDC is implemented so that for each actual hardware input device, the DCOM server creates a single input device instance and connects to the hardware device. When an application program connects with the input device control – which is an executable DCOM server - the DCOM server creates a MIIDC instance (and an interface) through which the application program communicates with the single input device instance. Data is provided for output by the single input device instance for each instance of the input device control, thus allowing simultaneous multiple applications to communicate with a single input device. Global settings are (MIIDC) instance specific. Additionally, the input device instance is protected so that multiple instances of the input device control program cannot perform tasks that would interfere with processing in another instance. Using this new approach, applications can be written which do not need

to account for the presence of another application possibly already using the same input device.

Other aspects of the present invention are directed towards the client-side mechanisms that enable an application program to communicate with the input device server executable. As described above, the MIIDC executable is implemented under a client-server architecture, where each application program is a client. Naturally, a client must be able to communicate with the server. The method of the present invention provides several mechanisms that enable an application program to communicate with the MIIDC server. In a PC/Windows environment, a first client-side mechanism is delivered via an ActiveX control called an input device portal. A second client-side mechanism also under a PC/Windows environment, is delivered via a DirectShow™ video capture source filter.

The client side mechanisms under the portal approach include communicating with the MIIDC server and supplying user-interface elements to an application. With the portal approach, all functionality of virtualizing an input device is performed by the MIIDC server, and thus, application programs communicating with the MIIDC server will require user-interface programming. To accomplish this, under the video-portal approach, a template is provided to allow various application program providers to generate their own custom input-device portal.

The client-side mechanism under the second approach (i.e. DirectShow approach) takes advantage of the standardized DirectShow modular components called filters. This second client-side mechanism replaces the standard source (media input) filter with a virtual source filter, which communicated directly with the MIIDC server. The virtual source filter is a client to the MIIDC server. With this mechanism, a DirectShow application cannot distinguish between the "real" and the "virtual" source filter. The advantage of this second client-side mechanism is that any application program written to function in a DirectShow environment, will be able to readily share an input device without the need for any additional user-interface programming before being able to communicate with the MIIDC server.

For a further understanding of the nature and advantages of the present invention, reference should be made to the following description in conjunction with the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

Fig. 1 is a block diagram showing the prior art method of a single application program communicating with a single video camera device.

Fig. 2 is a block diagram depicting one embodiment of the present multi-instance input device control program.

Fig. 3 is a flow chart showing the steps involved in an application connecting to a single input device.

DESCRIPTION OF THE SPECIFIC EMBODIMENTS

Fig. 2 shows a block diagram depicting one embodiment of the present multi-instance input device control program (MIIDC) in a PC/Windows environment. In this embodiment, the input device is a video camera, and the executable program is a DCOM executable server. This figure shows how multiple application programs may share a single video camera. Once a first application program 100 calls to connect to the video camera 108, the call is passed to the DCOM application program interface (API) 102. The appropriate Microsoft documentation or the Microsoft website may be referred to for a more detailed description of DCOM. The DCOM API 102 handles the loading of the DCOM executable program and establishes a connection from the application program to the DCOM executable program 200. The DCOM server 200 creates a single video camera instance 106 and a first MIIDC instance 104. Next, the DCOM server 200 connects the single video camera instance 106 to the video camera driver 107, the video camera driver 107 to the video camera device 108 and the first MIIDC instance 104 with the single video camera instance 106. The video camera instance 106 is a virtual interface to the physical video camera device 108. An instance is an actual usage and the virtual creation of a copy of an entity loaded into memory. In this embodiment all instance memory is allocated in the executable server. Finally the connection 300 is established allowing client application 100 to interact through the newly instantiated DCOM interface (single video camera instance) 106 with the video camera device 108.

Once a second application program 110 calls to connect to the video camera 108, the DCOM server 200 creates a second MIIDC instance 114, and connects it to the single video camera instance 106 thus allowing the second client application 110 to interact through the single video camera instance 106 with the video camera device 108 via the second established connection 310. Subsequent application program calls 120, et. seq. also interact through the DCOM instantiated single video camera instance interface

106 with the video camera device 108 via the subsequently established connections 320, et seq.

Figures 3 is a flowchart depicting the process of Fig. 2. Once a client application program calls to connect to the video camera device (step 103), the application program's call is sent to the DCOM API (step 203). Next, the DCOM API determines whether the DCOM implemented MIIDC executable is loaded or not. Typically the first client application program causes the MIIDC executable to be loaded. If the MIIDC executable server is not loaded, the DCOM API takes the call and causes the DCOM server to load the DCOM implemented MIIDC executable server (step 403). Next, the MIIDC server creates an input device control instance (step 503). If the MIIDC executable server had already been loaded, step 403 becomes unnecessary, and the next step after step 303 would be step 503. The MIIDC server next creates a single video camera instance and connects it to the video camera device, and connects the input device control instance to the single video camera instance (step 603). Finally, the MIIDC server creates an interface through which the first client application program communicates with the single camera instance (step 703).

The video camera instance 106 depicted on Fig. 2 is an interface with the video camera device that maintains the state of the input device control's instance. The input device instance 106 is a block of memory that maintains the necessary accounting of the number of connections that have been established with the video camera device, and the particular states of each of these connections. The video camera instance 106 also incorporates the logic necessary to prioritize the requests from each input device control instance connection and multiplex and resolve conflicting requests. Since the MIIDC server exists as a separate process, video (and audio) data must be replicated for each client requiring access to the video (and audio) data. To reduce data replication, the MIIDC server is designed to record video (and audio), detect motion, save pictures, as well as other functions which are typical of a media source capture device. The MIIDC server thus limits the data replication to only those applications requiring direct access to media source (e.g., video and audio) data.

For example, the first input device instance may be requesting a video stream having a resolution of 640 by 480 pixels, while the second and third instances may be requesting video streams having 320 by 480 and 160 by 120 pixel resolutions respectively. In such a scenario, the video camera instance 106 would then decide to capture video at the largest resolution of 640 by 480 pixels and then scale it or crop it

down to the lower resolutions being requested by the second and third instances. Following the same logic, if consequently the first video instance disconnects from the video camera, the video camera instance 106, would then resolve the requests from the second and third instances requesting 320 by 480 and 160 by 120 pixel resolutions respectively, by capturing video at the highest requested resolution of 320 by 480 pixels to satisfy the second instance's request and then scaling down or cropping the 320 by 480 pixels video stream down to 160 by 120 pixels to satisfy the third instance's request.

In another example involving three input device control instances, the first input device control instance may be sending a motion detection command to the virtual video camera device, while the other two instances are only requesting video streams. Now the video camera instance 106 would capture video at the highest demanded resolution and only pass that video stream through a motion detection calculation for the first input device control instance.

In yet another example involving three input device control instances, the second input device control instance may be requesting a text overlay on the video image, while the other two instances are only requesting video stream captures. Now, the video camera instance 106 would capture video at the highest demanded and only add the text overlay to the stream flowing to the second input device request.

While the embodiments described thus far were generally described in the context of a video camera that is interfaced with a personal computer host, the scope of the present invention is not meant to be limited solely to a video camera or even a particular type of personal computer host. As described above, the embodiments of the present invention are directed towards the simultaneous sharing of an input device by several application programs by virtualizing a device driver file which is in turn achieved by implementing the input device control program as an executable server. While the input device described above is a video camera, another input device that can be configured to be simultaneously shared is a microphone. Thus, the input device instance (106 on Fig. 2) incorporates the logic necessary to prioritize the requests from each input device control instance and multiplex and resolve conflicting requests. Extending the sharing capabilities of video source to also include an audio input source, is not only a natural one, but it is also almost mandatory, since video and audio are most commonly bundled together as naturally complementary media sources.

For example, referring back to Fig. 2, it is expected to consider that a microphone (not shown) is also recording sound while the device 108 is recording video.

Then for example, the first input device instance may be requesting audio having a bit depth of 16-bits at 44.1 kHz, while the second instance may be requesting audio streams having an 8-bit depth at 11.025 kHz. In such a scenario, the input device instance will then decide to capture audio at the highest sampling rate and bit depth and then scale, or compress it down to the lower bit depth or sampling rate being requested by the second instance.

The MIIDC and the method of virtualizing an input device are implementable on many computing platforms running various operating systems. A media source input device such as a video camera or a microphone is commonly interfaced with a host computer. The host computer is most commonly a personal computer, such as the commonly available PC of Mac computers. However, since advancements in technology are blurring the boundaries between computing and communication devices, a host computer as used herein is synonymous with an intelligent host, and an intelligent host as used herein is meant to include other examples of any host having a processor, memory, means for input and output, and means for storage. Other examples of intelligent hosts, which are also equally qualified to be used in conjunction with embodiments of the present invention include a handheld computer, an interactive set-top box, a thin client computing device, a personal access device, a personal digital assistants, and an internet appliance.

Other aspects of the present invention are directed towards the client-side mechanisms that enable an application program to communicate with the input device server executable. As described above, the MIIDC executable is implemented under a client-server architecture, where each application program is a client. Therefore, a client must be able to communicate with the server. The method of the present invention provides several mechanisms that enable an application program to communicate with the MIIDC server. In a PC/Windows environment, a first client-side mechanism is delivered via an ActiveX control called an input device portal. A second client-side mechanism also under a PC/Windows environment, is delivered via a DirectShow video capture source filter.

The client side mechanisms under the portal approach include communicating with the MIIDC server and supplying user-interface elements to an application. With the portal approach, all functionality of virtualizing an input device is performed by the MIIDC server, and thus, application programs communicating with the MIIDC server will require user-interface programming. To accomplish this, under the

video-portal approach, a template is provided to allow various application program providers to generate their own custom input-device portal.

The client-side mechanism under the second approach (i.e. DirectShow approach) takes advantage of the standardized DirectShow modular components called filters. DirectShow™ services from Microsoft™ provide playback services for multimedia streams including capture of multimedia streams from devices. At the heart of the DirectShow™ services is a modular system of pluggable components called filters. These modular components can be classified as a source, transform or renderer. Filters operate on data streams by reading, copying, modifying or writing the data to a file or rendering the file to an output device. The filters have input and output means and are connected to each other in a configuration called a filter graph. Application programs use an object called the filter graph manager to assemble the filter graph and move data through it. The filter graph manager handles the data flow from an input device to the playback device. A further description of DirectShow™ services and the Microsoft™ DirectX™ media software development kit can be obtained by referring to appropriate documentation as is known to those of skill in the art.

This second client-side mechanism replaces the standard source (media input) filter with a virtual source filter, which communicates directly with the MIIDC server. The virtual source filter is a client to the MIIDC server. With this mechanism, a DirectShow application cannot distinguish between the "real" and the "virtual" source filter. The advantage of this second client-side mechanism is that any application program written to function in a DirectShow environment, will be able to readily share an input device without the need for any additional user-interface programming before being able to communicate with the MIIDC server.

As will be understood by those skilled in the art, the present invention may be embodied in other specific forms without departing from the essential characteristics thereof. For example, the MIIDC could be implemented as any other executable process other than a DCOM based process and any other interfacing protocol other than the DCOM interface could be used to allow multiple application programs to communicate with that process. These other embodiments are intended to be included within the scope of the present invention, which is set froth in the following claims.